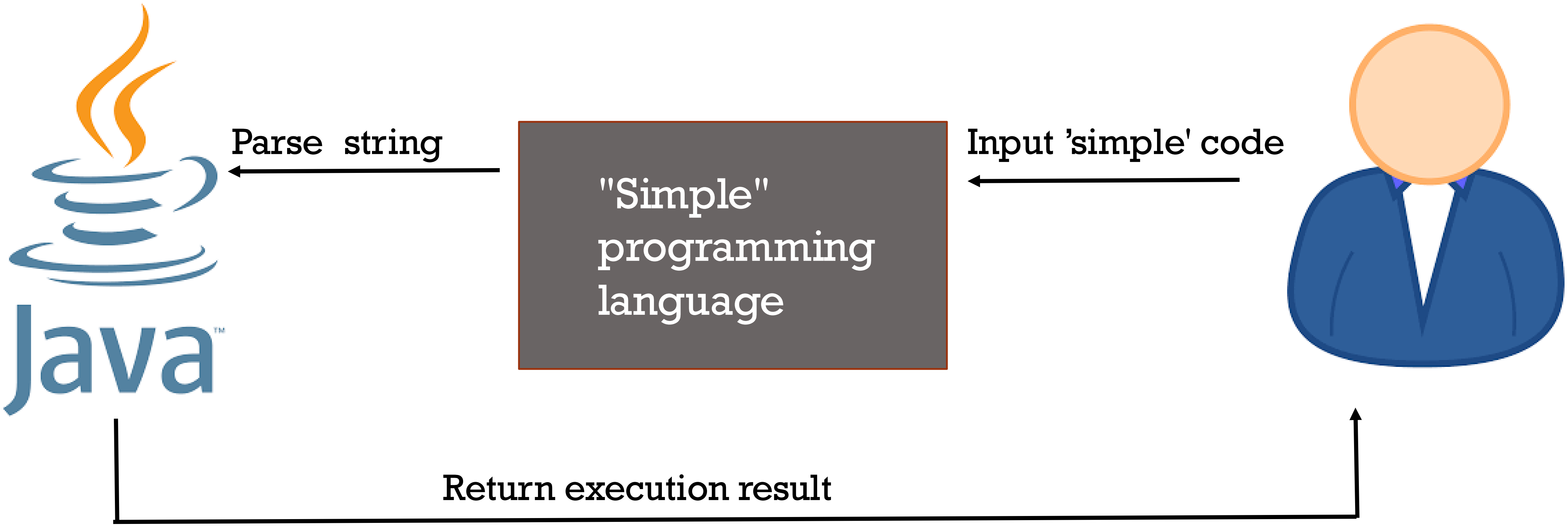
COMP2021 GROUP PRESENTATION

CONTENT

- Overall architecture
- Program design choice
- Usage of Object-Orientation

USER INTERACTION ARCHITECTURE



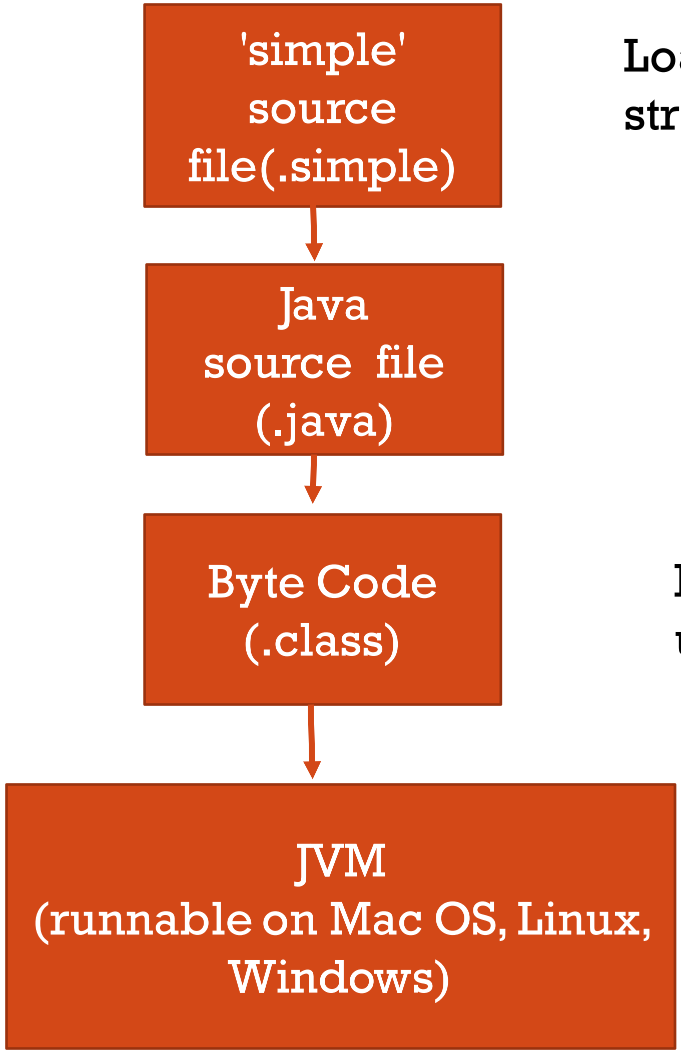
Parse string

"Simple" programming language

Input 'simple' code

Return execution result

EXECUTION ARCHITECTURE



'simple' source file(.simple)

Java source file (.java)

Byte Code (.class)

JVM (runnable on Mac OS, Linux, Windows)

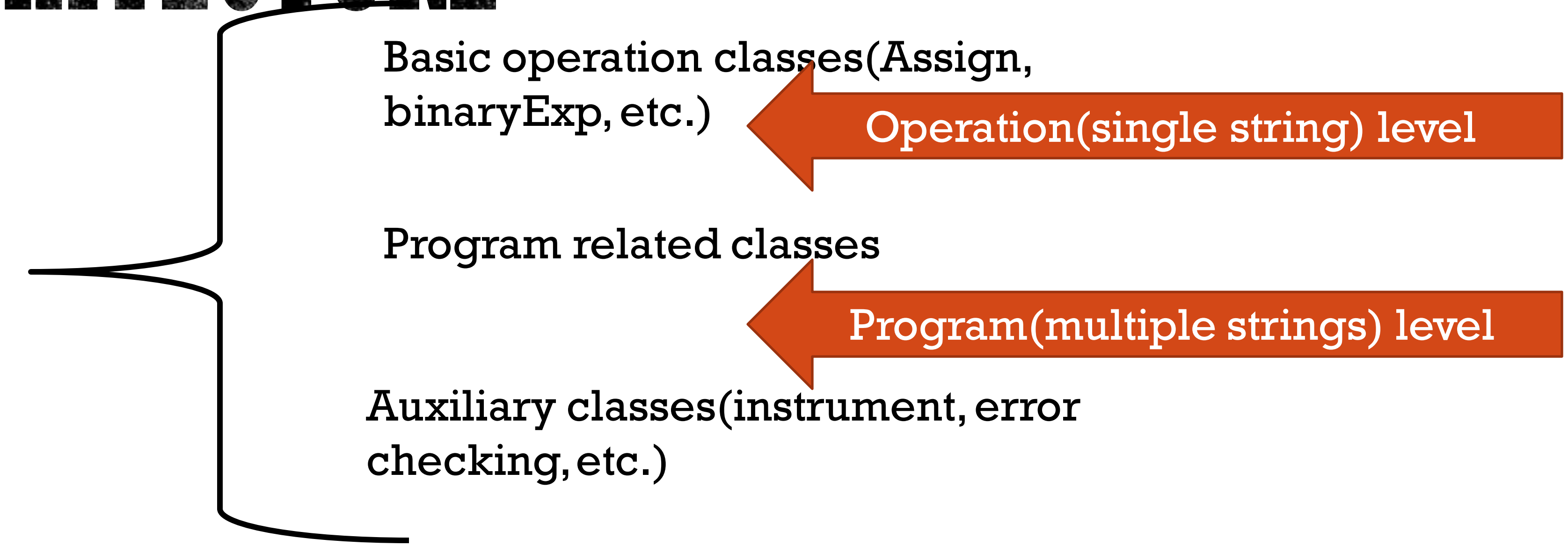
Load/input 'simple' language as string in java environment

Create simple text file with .java extension

Now 'simple' code is understandable by JVM

PROJECT ARCHITECTURE

- Package -> model



Basic operation classes(Assign, binaryExp, etc.)

Operation(single string) level

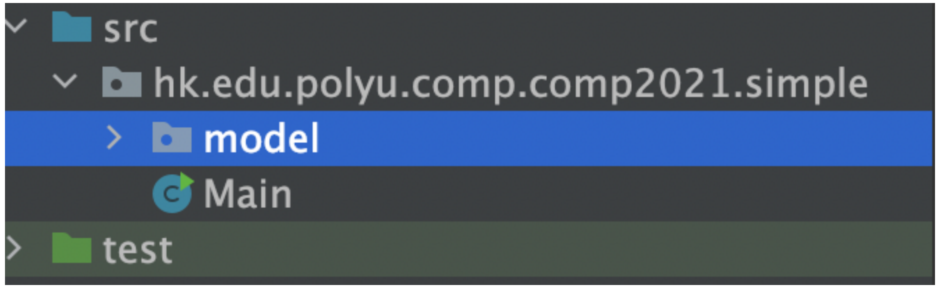
Program related classes

Program(multiple strings) level

Auxiliary classes(instrument, error checking, etc.)

- Class: main

- Test



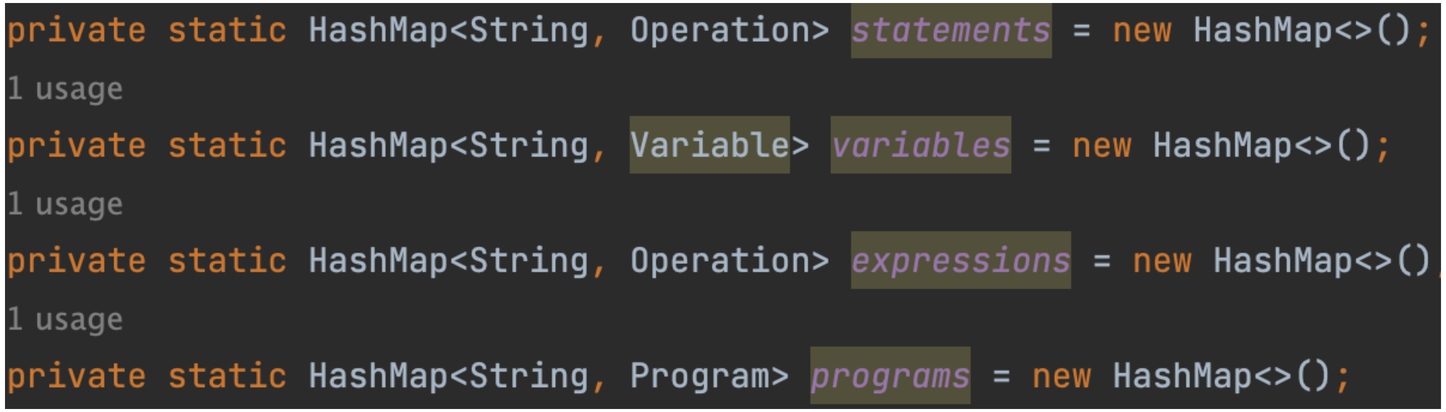
```
src
  hk.edu.polyu.comp.comp2021.simple
    model
    Main
test
```

BASIC IDEA OF PROJECT

- As all classes we need to retrieve can be divided into four levels…
  - (statements, variables, expressions, program)

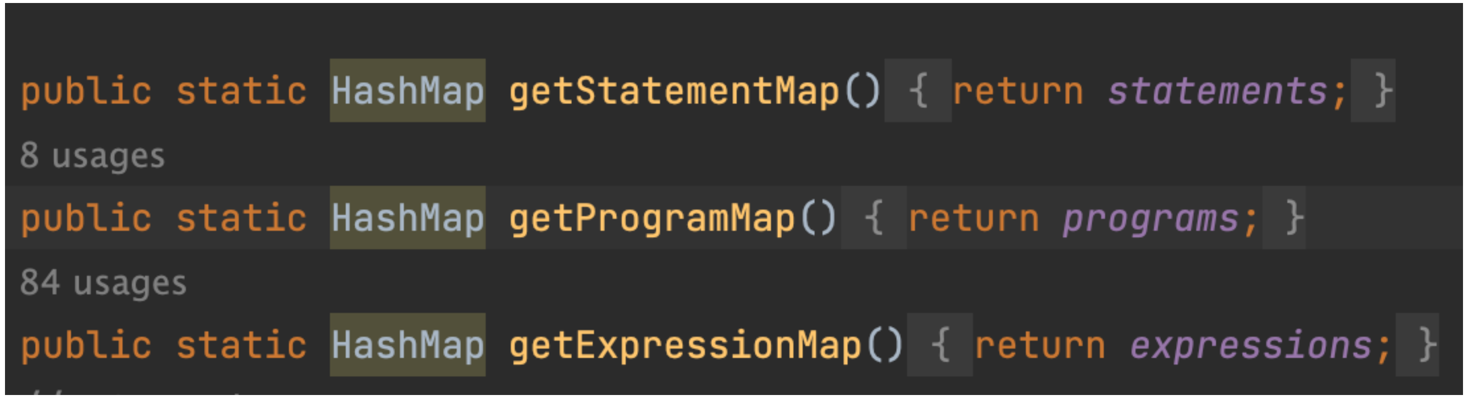
- Use **hashmap** to store the classes
  - Easy to retrieve
  - Save space
  - Map operation labels with operation instances

BASIC IDEA OF PROJECT



```
private static HashMap<String, Operation> statements = new HashMap<>();
1 usage
private static HashMap<String, Variable> variables = new HashMap<>();
1 usage
private static HashMap<String, Operation> expressions = new HashMap<>()
1 usage
private static HashMap<String, Program> programs = new HashMap<>();
```

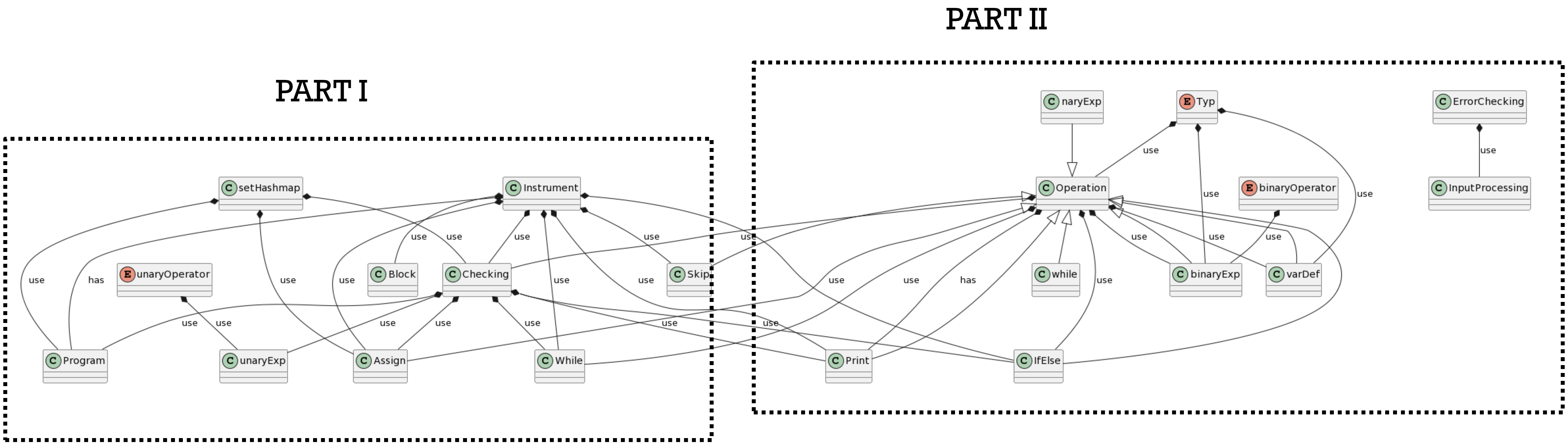
Store operation/ program



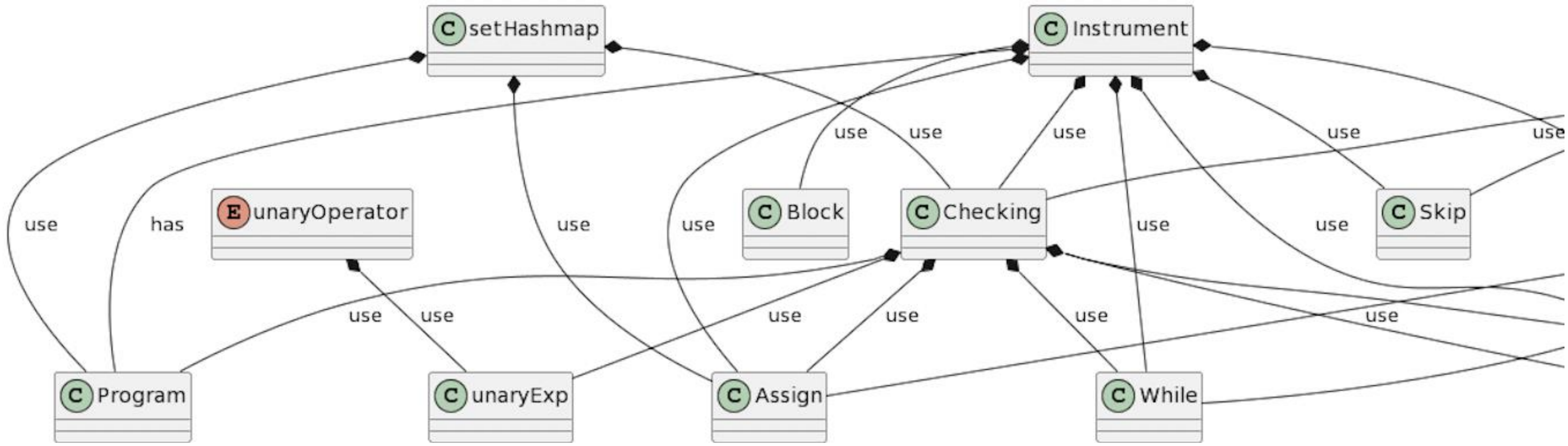
```
public static HashMap getStatementMap() { return statements; }
8 usages
public static HashMap getProgramMap() { return programs; }
84 usages
public static HashMap getExpressionMap() { return expressions; }
```

Retrieve the corresponding hashmap

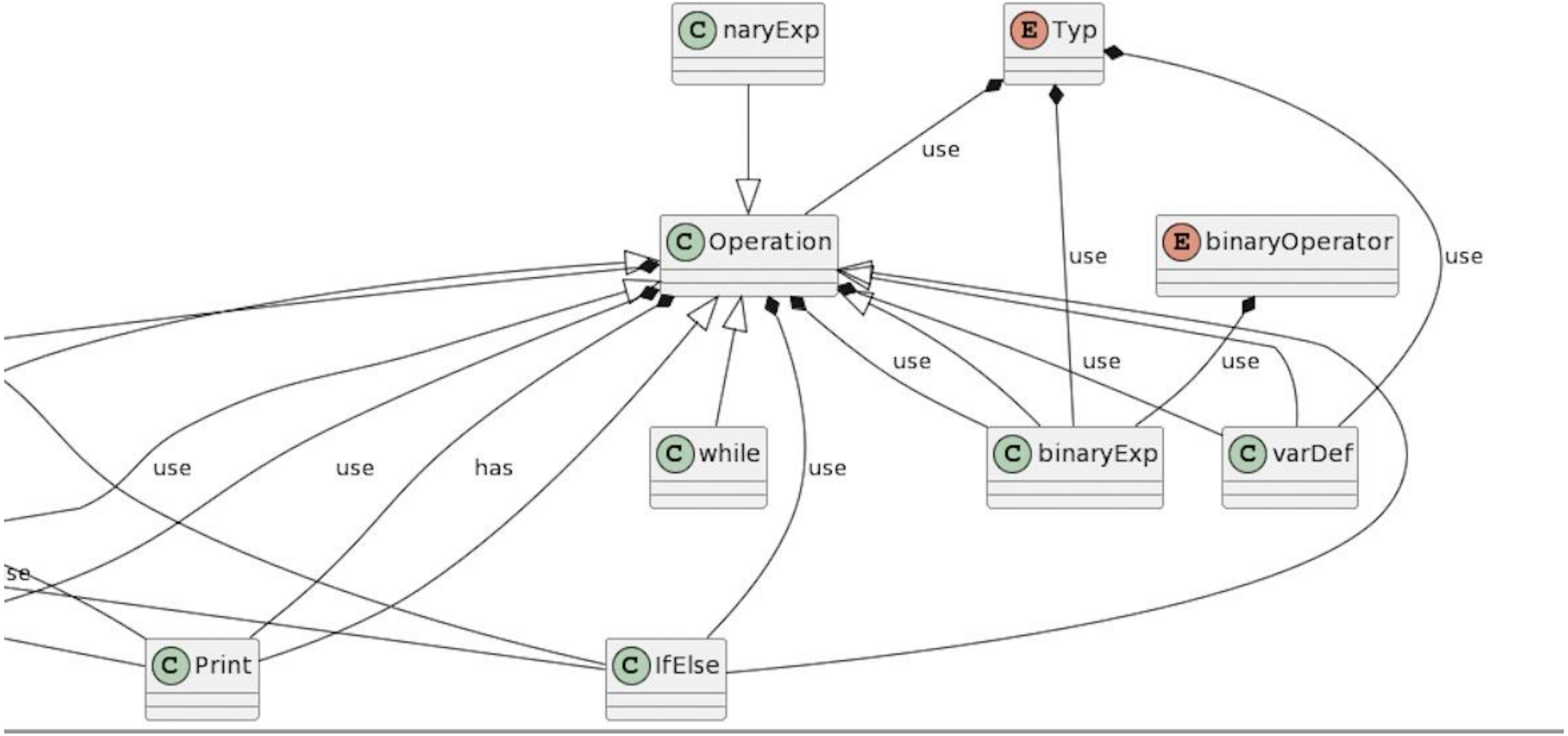
UML DIAGRAM

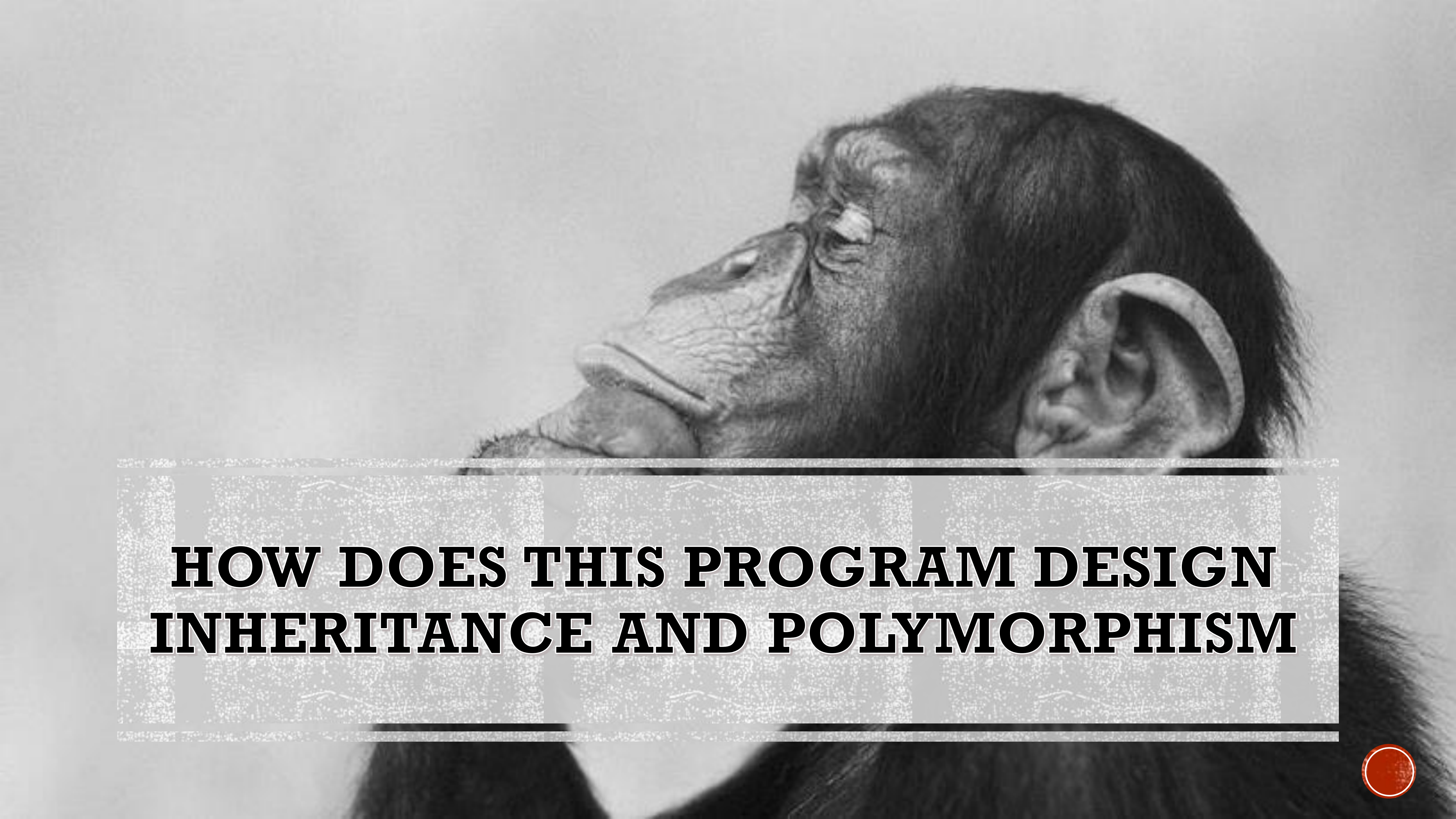

PART II
PART I
naryExp
Typ
ErrorChecking
setHashmap
Instrument
Operation
binaryOperator
InputProcessing
unaryOperator
Block
Checking
Skip
while
binaryExp
varDef
Program
unaryExp
Assign
While
Print
IfElse
use
has

UML DIAGRAM (PART I)



UML DIAGRAM (PART II)



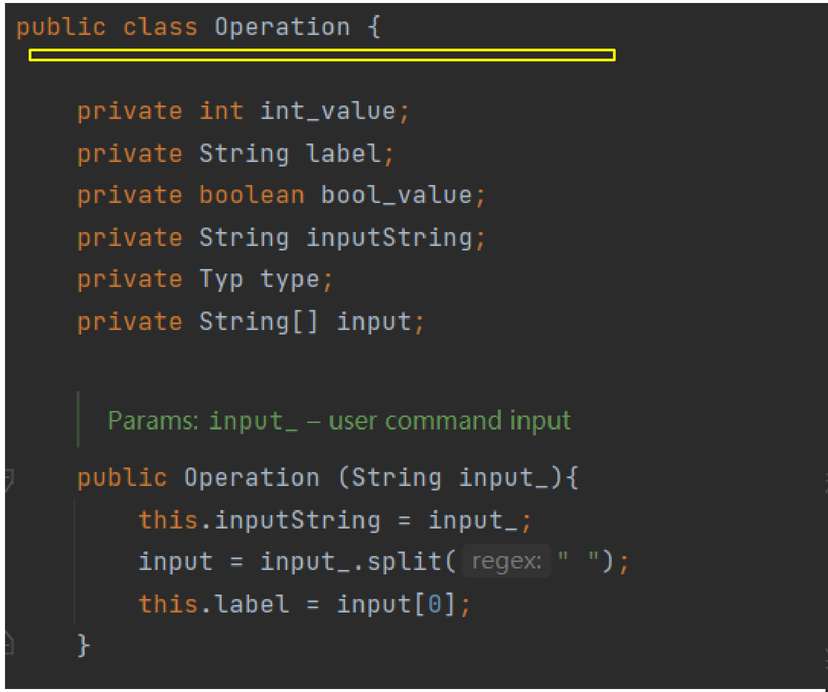


HOW DOES THIS PROGRAM DESIGN INHERITANCE AND POLYMORPHISM

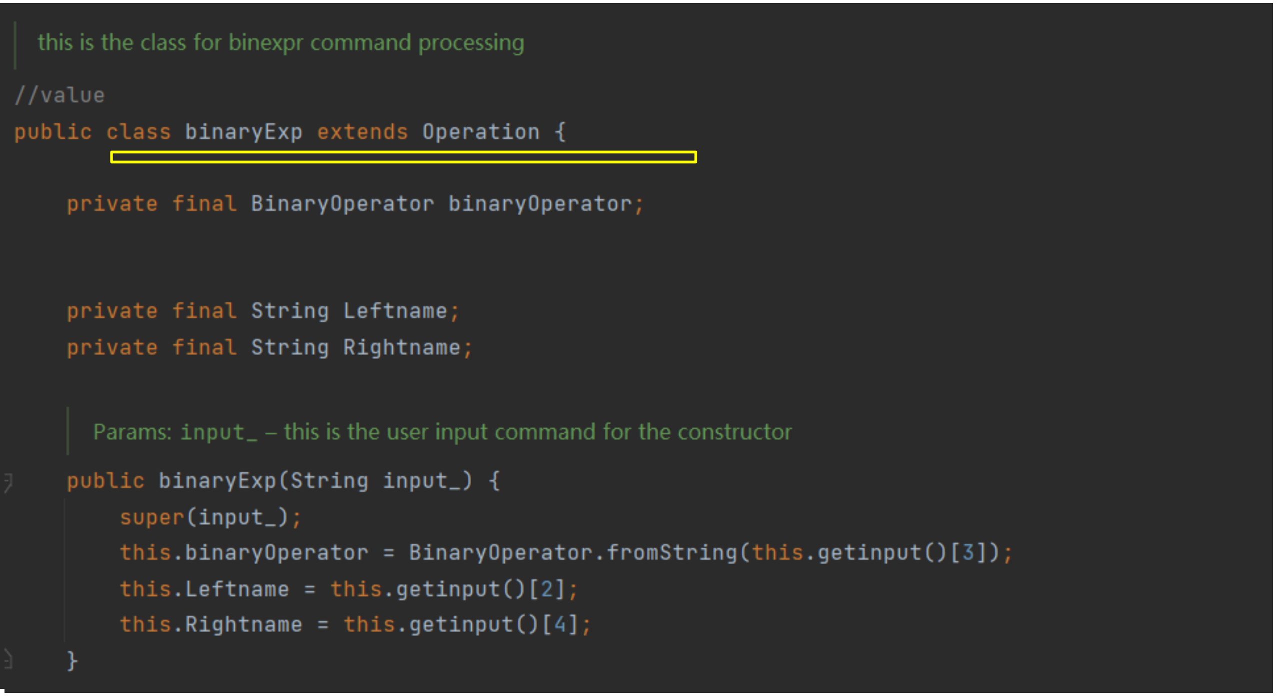
WHAT IS INHERITANCE AND POLYMORPHISM

- Inheritance -
- Inheriting members from an existing class to define a new class
- Add new members or
- Redefining or replacing existing members

- Polymorphism -
- Provide different implementations of a method, depending on the type of object that is passed to the method.



```
public class Operation {

    private int int_value;
    private String label;
    private boolean bool_value;
    private String inputString;
    private Typ type;
    private String[] input;

    Params: input_ – user command input
    public Operation (String input_){
        this.inputString = input_;
        input = input_.split( regex: " ");
        this.label = input[0];
    }
```

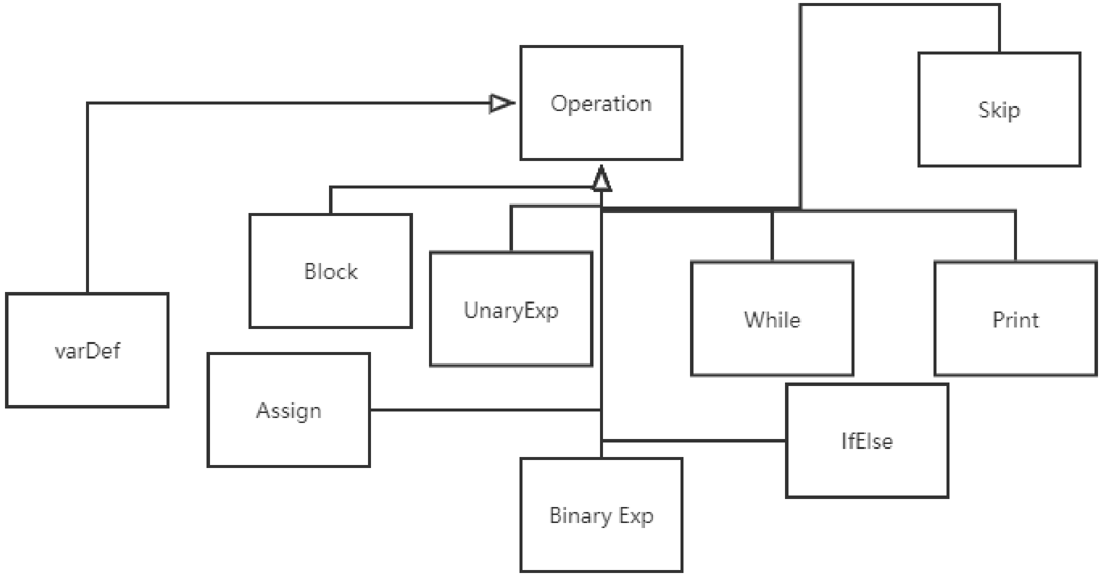


```
this is the class for binexpr command processing
//value
public class binaryExp extends Operation {

    private final BinaryOperator binaryOperator;


    private final String Leftname;
    private final String Rightname;

    Params: input_ – this is the user input command for the constructor
    public binaryExp(String input_) {
        super(input_);
        this.binaryOperator = BinaryOperator.fromString(this.getinput()[3]);
        this.Leftname = this.getinput()[2];
        this.Rightname = this.getinput()[4];
    }
```

INHERITANCE

- The Operation Class is created to process the input value to get type, label, input String, and so on.
- Write a Program in general fashion.
- Software Reusability

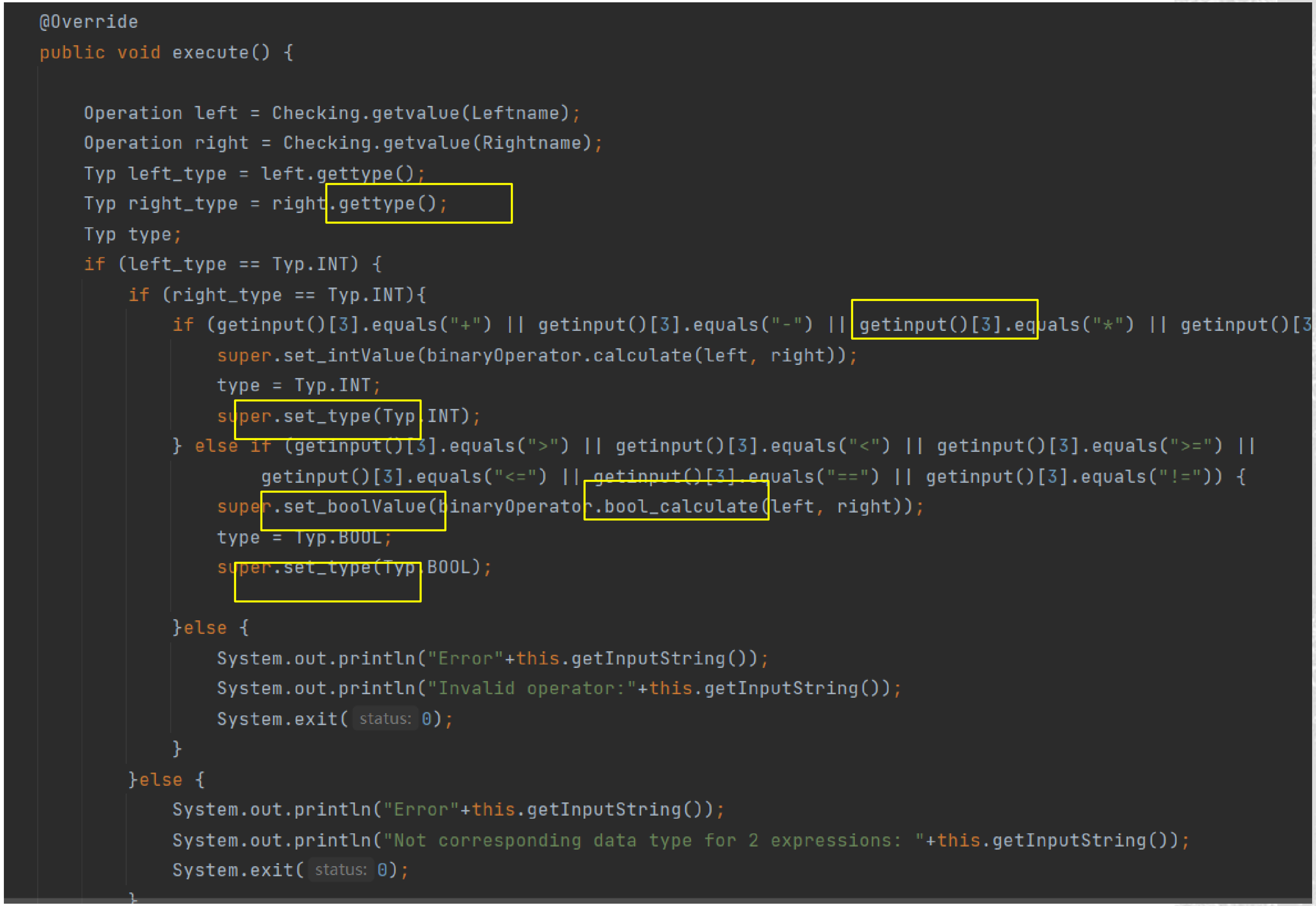
OPERATION



- General Concept:
- Process the Input

- Specified Concept:
- Achieve Block, varDef, and other specified Requirement.

AS AN EXAMPLE

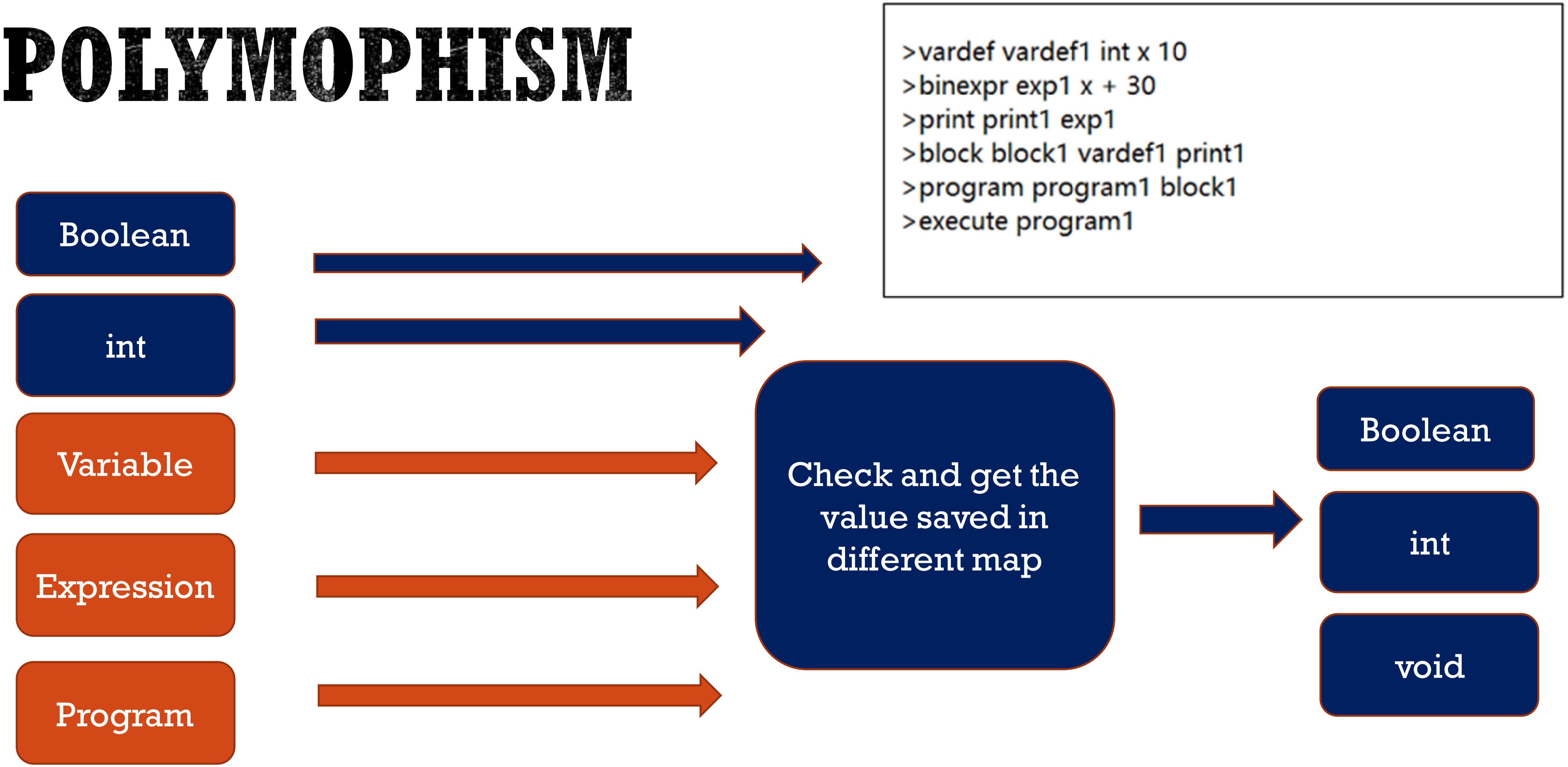


```java
@Override
public void execute() {

    Operation left = Checking.getvalue(Leftname);
    Operation right = Checking.getvalue(Rightname);
    Typ left_type = left.gettype();
    Typ right_type = right.gettype();
    Typ type;
    if (left_type == Typ.INT) {
        if (right_type == Typ.INT){
            if (getinput()[3].equals("+") || getinput()[3].equals("-") || getinput()[3].equals("*") || getinput()[3
                super.set_intValue(binaryOperator.calculate(left, right));
                type = Typ.INT;
                super.set_type(Typ.INT);
            } else if (getinput()[3].equals(">") || getinput()[3].equals("<") || getinput()[3].equals(">=") ||
                    getinput()[3].equals("<=") || getinput()[3].equals("==") || getinput()[3].equals("!=")) {
                super.set_boolValue(binaryOperator.bool_calculate(left, right));
                type = Typ.BOOL;
                super.set_type(Typ.BOOL);

            }else {
                System.out.println("Error"+this.getInputString());
                System.out.println("Invalid operator:"+this.getInputString());
                System.exit( status: 0);
            }
        }else {
            System.out.println("Error"+this.getInputString());
            System.out.println("Not corresponding data type for 2 expressions: "+this.getInputString());
            System.exit( status: 0);
        }
```

- Binary expression has three processing possibilities:
- int – int, like + -
- Int - boolean, >
- Bool – bool, &&

- Using the function existed from operation, like get type(), getinput(), can make the processing process with higher understandability and efficiency

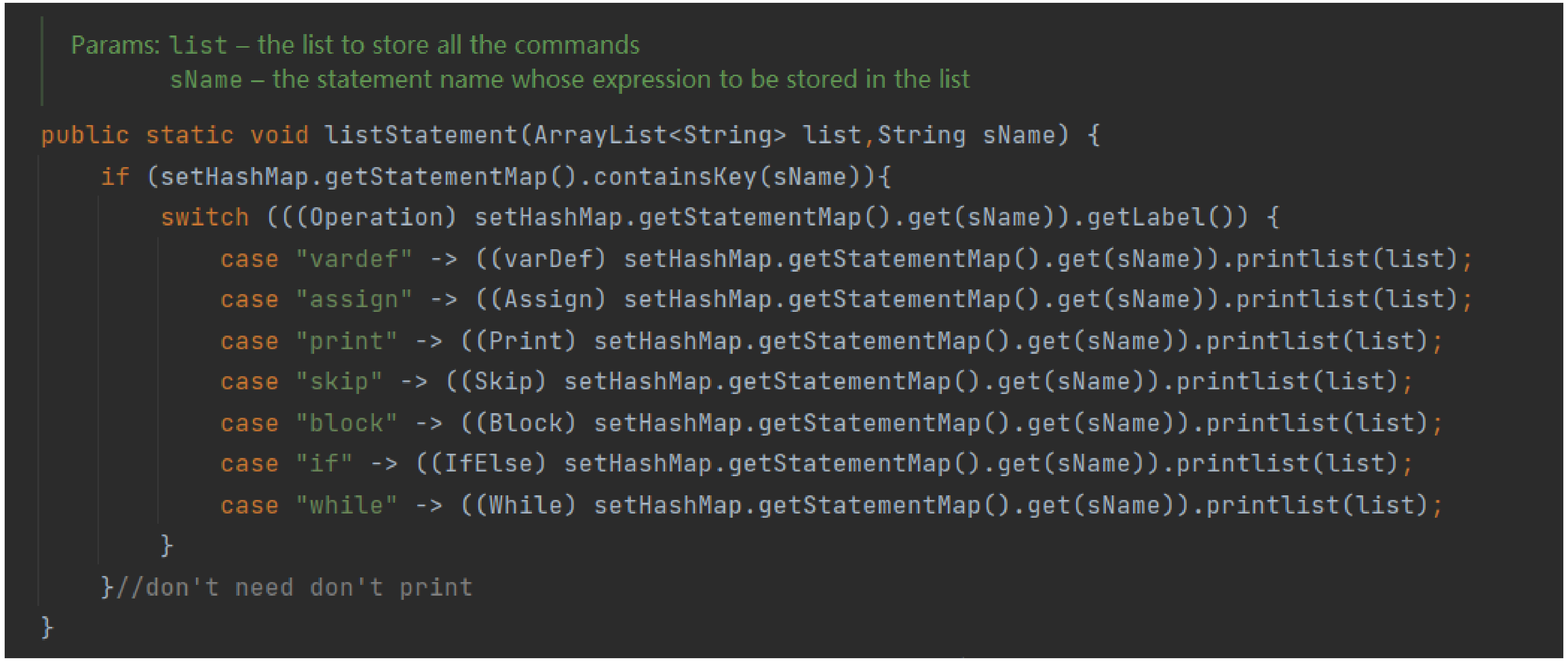
POLYMOPHISM



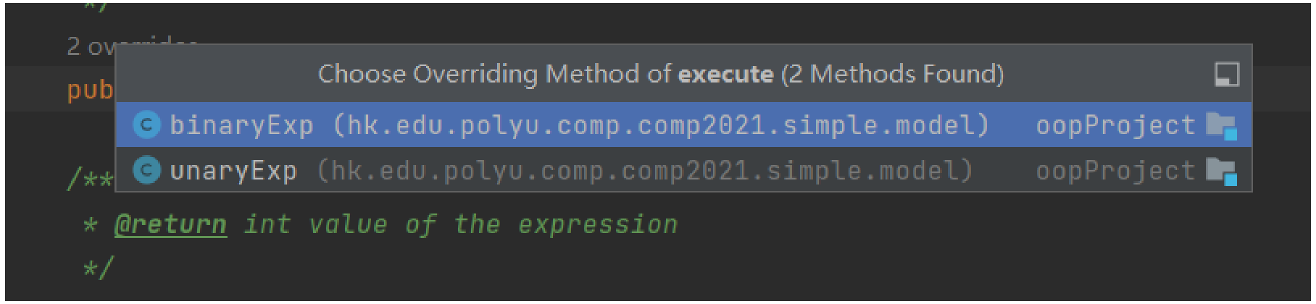
```
>vardef vardef1 int x 10
>binexpr exp1 x + 30
>print print1 exp1
>block block1 vardef1 print1
>program program1 block1
>execute program1
```

POLYMOPHISM

- The ability to express different functionality through a common interface



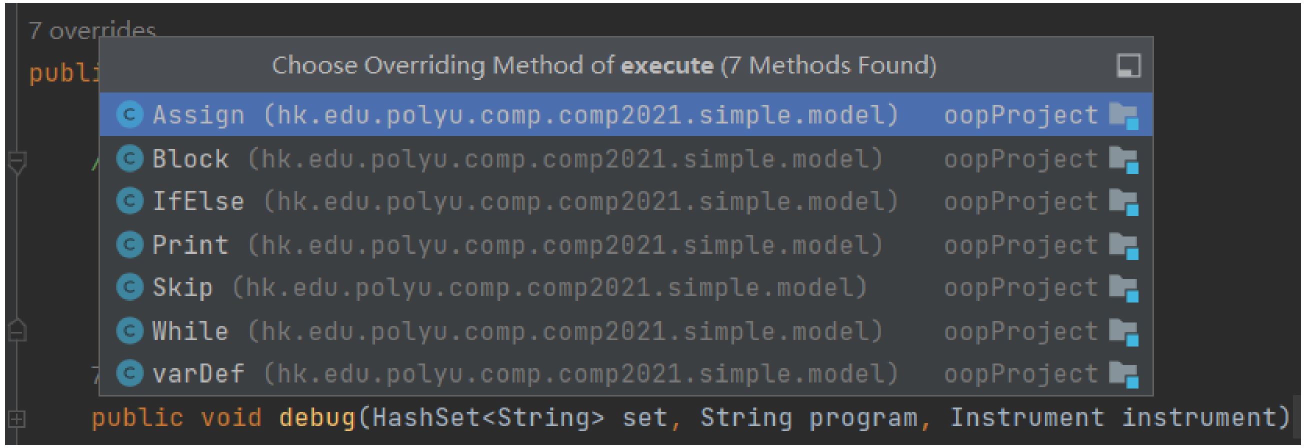
```
Params: list – the list to store all the commands
        sName – the statement name whose expression to be stored in the list
public static void listStatement(ArrayList<String> list,String sName) {
    if (setHashMap.getStatementMap().containsKey(sName)){
        switch (((Operation) setHashMap.getStatementMap().get(sName)).getLabel()) {
            case "vardef" -> ((varDef) setHashMap.getStatementMap().get(sName)).printlist(list);
            case "assign" -> ((Assign) setHashMap.getStatementMap().get(sName)).printlist(list);
            case "print" -> ((Print) setHashMap.getStatementMap().get(sName)).printlist(list);
            case "skip" -> ((Skip) setHashMap.getStatementMap().get(sName)).printlist(list);
            case "block" -> ((Block) setHashMap.getStatementMap().get(sName)).printlist(list);
            case "if" -> ((IfElse) setHashMap.getStatementMap().get(sName)).printlist(list);
            case "while" -> ((While) setHashMap.getStatementMap().get(sName)).printlist(list);
        }
    }//don't need don't print
}
```



```
  */
  2 ov
  pub

  /**
   * @return int value of the expression
   */
```

Choose Overriding Method of **execute** (2 Methods Found)

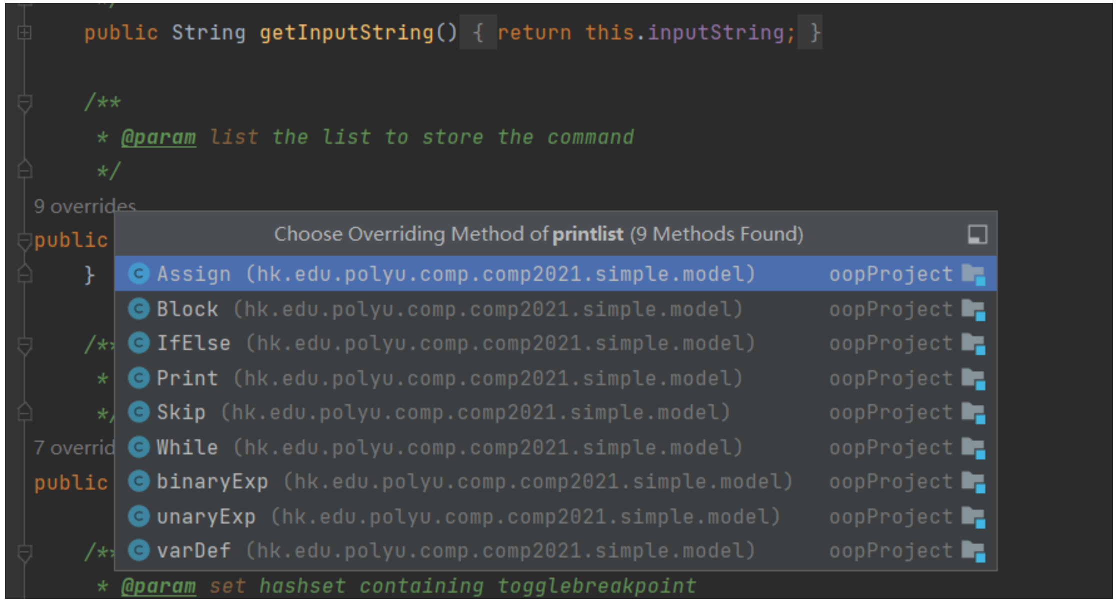
```
binaryExp (hk.edu.polyu.comp.comp2021.simple.model)   oopProject
unaryExp (hk.edu.polyu.comp.comp2021.simple.model)    oopProject
```



```
7 overrides
publi

    /
    7
    public void debug(HashSet<String> set, String program, Instrument instrument)
```

Choose Overriding Method of **execute** (7 Methods Found)

```
Assign (hk.edu.polyu.comp.comp2021.simple.model)   oopProject
Block (hk.edu.polyu.comp.comp2021.simple.model)    oopProject
IfElse (hk.edu.polyu.comp.comp2021.simple.model)   oopProject
Print (hk.edu.polyu.comp.comp2021.simple.model)    oopProject
Skip (hk.edu.polyu.comp.comp2021.simple.model)     oopProject
While (hk.edu.polyu.comp.comp2021.simple.model)    oopProject
varDef (hk.edu.polyu.comp.comp2021.simple.model)   oopProject
```



```
    public String getInputString() { return this.inputString; }

    /**
     * @param list the list to store the command
     */
9 overrides
public
    }

    /*
     *
     */
7 overrid
public

    /*
     * @param set hashset containing togglebreakpoint
```

Choose Overriding Method of **printlist** (9 Methods Found)

```
Assign (hk.edu.polyu.comp.comp2021.simple.model)      oopProject
Block (hk.edu.polyu.comp.comp2021.simple.model)       oopProject
IfElse (hk.edu.polyu.comp.comp2021.simple.model)      oopProject
Print (hk.edu.polyu.comp.comp2021.simple.model)       oopProject
Skip (hk.edu.polyu.comp.comp2021.simple.model)        oopProject
While (hk.edu.polyu.comp.comp2021.simple.model)       oopProject
binaryExp (hk.edu.polyu.comp.comp2021.simple.model)   oopProject
unaryExp (hk.edu.polyu.comp.comp2021.simple.model)    oopProject
varDef (hk.edu.polyu.comp.comp2021.simple.model)      oopProject
```

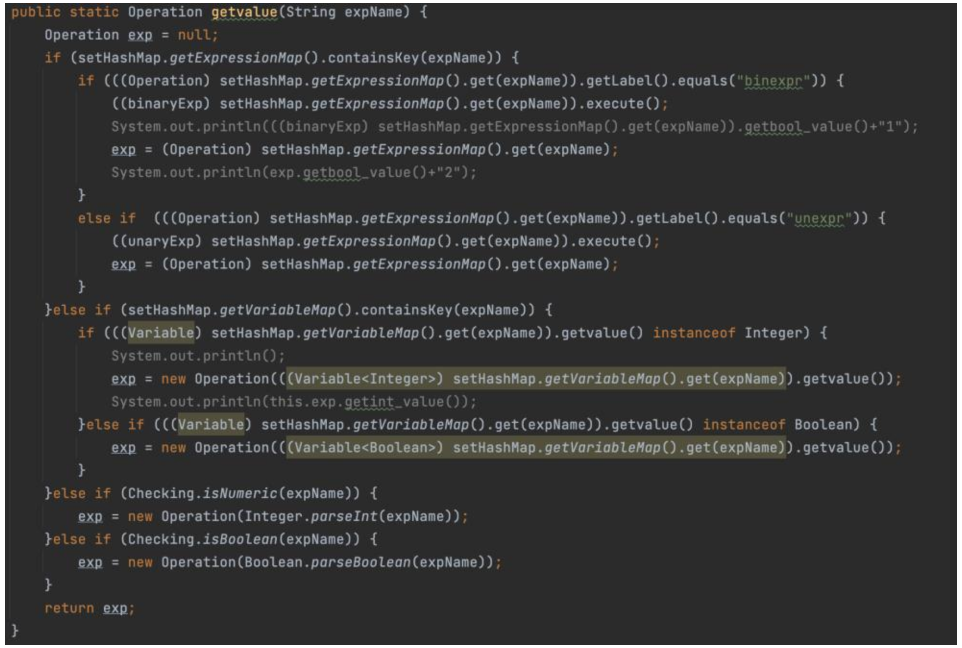
OVERRIDE THE METHOD OF SUPER CLASS

WHAT DOSE THE OOP MAKE DIFFERENCE?

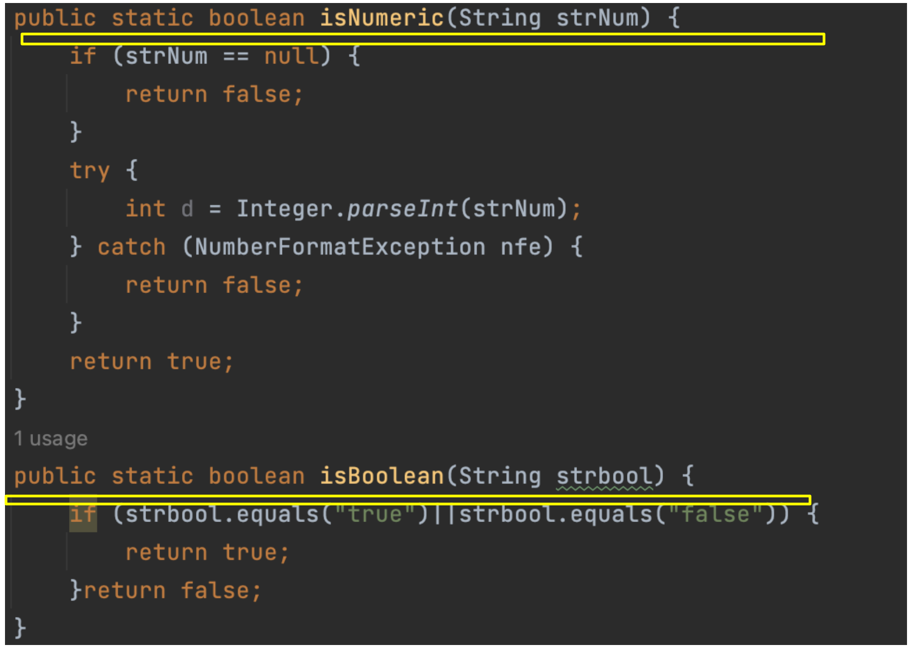
RECALL THE DEFINITION

- Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

- The organization of an object-oriented program also makes the method beneficial to collaborative development, where projects are divided into groups. Additional benefits of OOP include code reusability, scalability and efficiency.



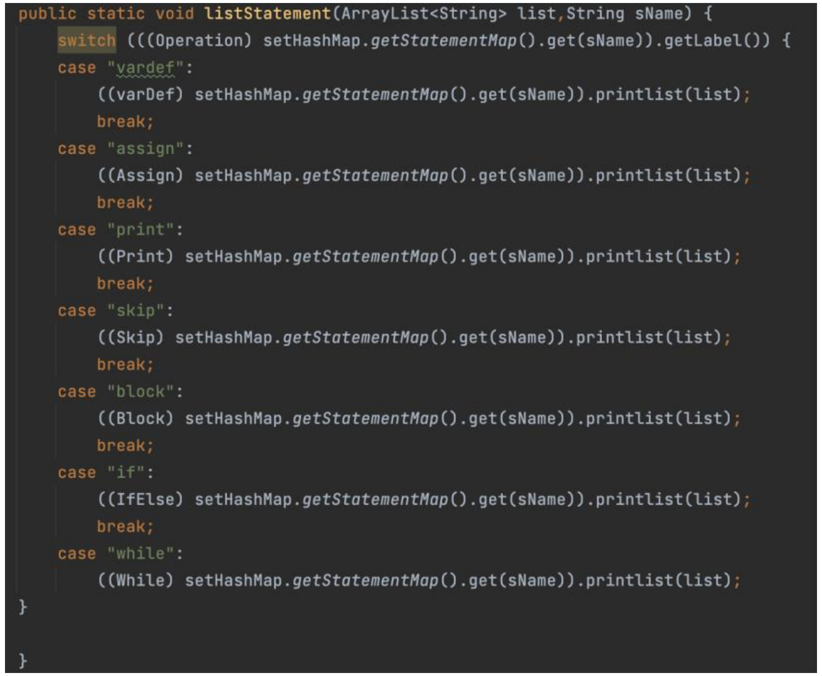
```java
public static Operation getvalue(String expName) {
    Operation exp = null;
    if (setHashMap.getExpressionMap().containsKey(expName)) {
        if (((Operation) setHashMap.getExpressionMap().get(expName)).getLabel().equals("binexpr")) {
            ((binaryExp) setHashMap.getExpressionMap().get(expName)).execute();
            System.out.println(((binaryExp) setHashMap.getExpressionMap().get(expName)).getbool_value()+"1");
            exp = (Operation) setHashMap.getExpressionMap().get(expName);
            System.out.println(exp.getbool_value()+"2");
        }
        else if  (((Operation) setHashMap.getExpressionMap().get(expName)).getLabel().equals("unexpr")) {
            ((unaryExp) setHashMap.getExpressionMap().get(expName)).execute();
            exp = (Operation) setHashMap.getExpressionMap().get(expName);
        }
    }else if (setHashMap.getVariableMap().containsKey(expName)) {
        if (((Variable) setHashMap.getVariableMap().get(expName)).getvalue() instanceof Integer) {
            System.out.println();
            exp = new Operation(((Variable<Integer>) setHashMap.getVariableMap().get(expName)).getvalue());
            System.out.println(this.exp.getint_value());
        }else if (((Variable) setHashMap.getVariableMap().get(expName)).getvalue() instanceof Boolean) {
            exp = new Operation(((Variable<Boolean>) setHashMap.getVariableMap().get(expName)).getvalue());
        }
    }else if (Checking.isNumeric(expName)) {
        exp = new Operation(Integer.parseInt(expName));
    }else if (Checking.isBoolean(expName)) {
        exp = new Operation(Boolean.parseBoolean(expName));
    }
    return exp;
}
```



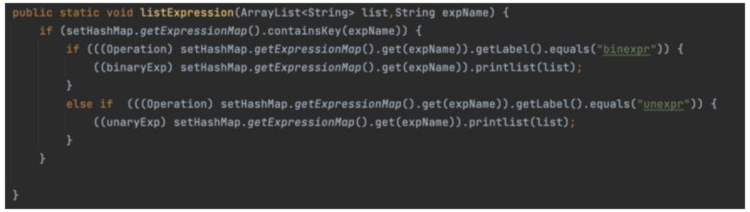
```java
public static boolean isNumeric(String strNum) {
    if (strNum == null) {
        return false;
    }
    try {
        int d = Integer.parseInt(strNum);
    } catch (NumberFormatException nfe) {
        return false;
    }
    return true;
}
1 usage
public static boolean isBoolean(String strbool) {
    if (strbool.equals("true")||strbool.equals("false")) {
        return true;
    }return false;
}
```

REUSABILITY

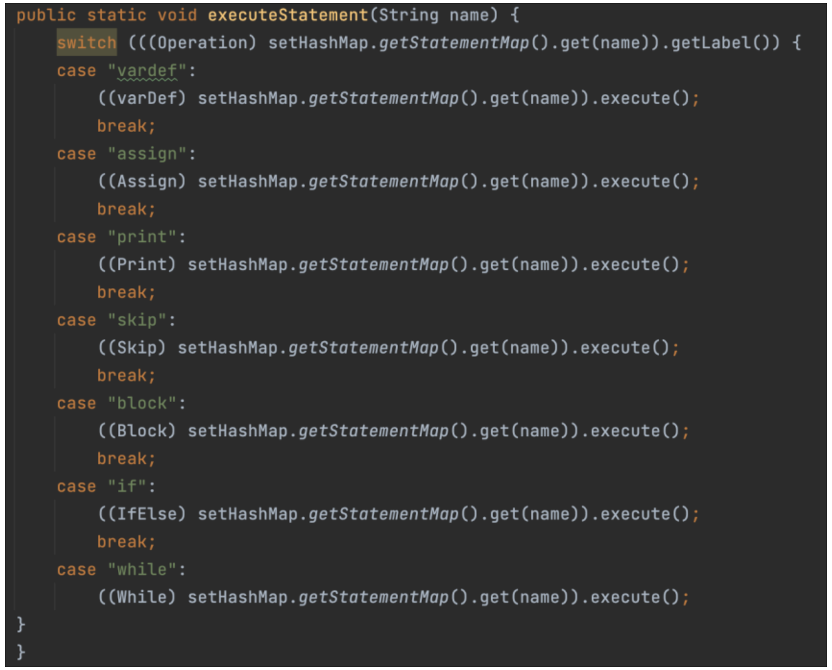
- We first find that there will be many "value-check" and "value-get" operation in achieving each small function.
- So we make it a separate class, and multiple call it when needed.



```
public static void listStatement(ArrayList<String> list,String sName) {
    switch (((Operation) setHashMap.getStatementMap().get(sName)).getLabel()) {
    case "vardef":
        ((varDef) setHashMap.getStatementMap().get(sName)).printlist(list);
        break;
    case "assign":
        ((Assign) setHashMap.getStatementMap().get(sName)).printlist(list);
        break;
    case "print":
        ((Print) setHashMap.getStatementMap().get(sName)).printlist(list);
        break;
    case "skip":
        ((Skip) setHashMap.getStatementMap().get(sName)).printlist(list);
        break;
    case "block":
        ((Block) setHashMap.getStatementMap().get(sName)).printlist(list);
        break;
    case "if":
        ((IfElse) setHashMap.getStatementMap().get(sName)).printlist(list);
        break;
    case "while":
        ((While) setHashMap.getStatementMap().get(sName)).printlist(list);
}

}
```



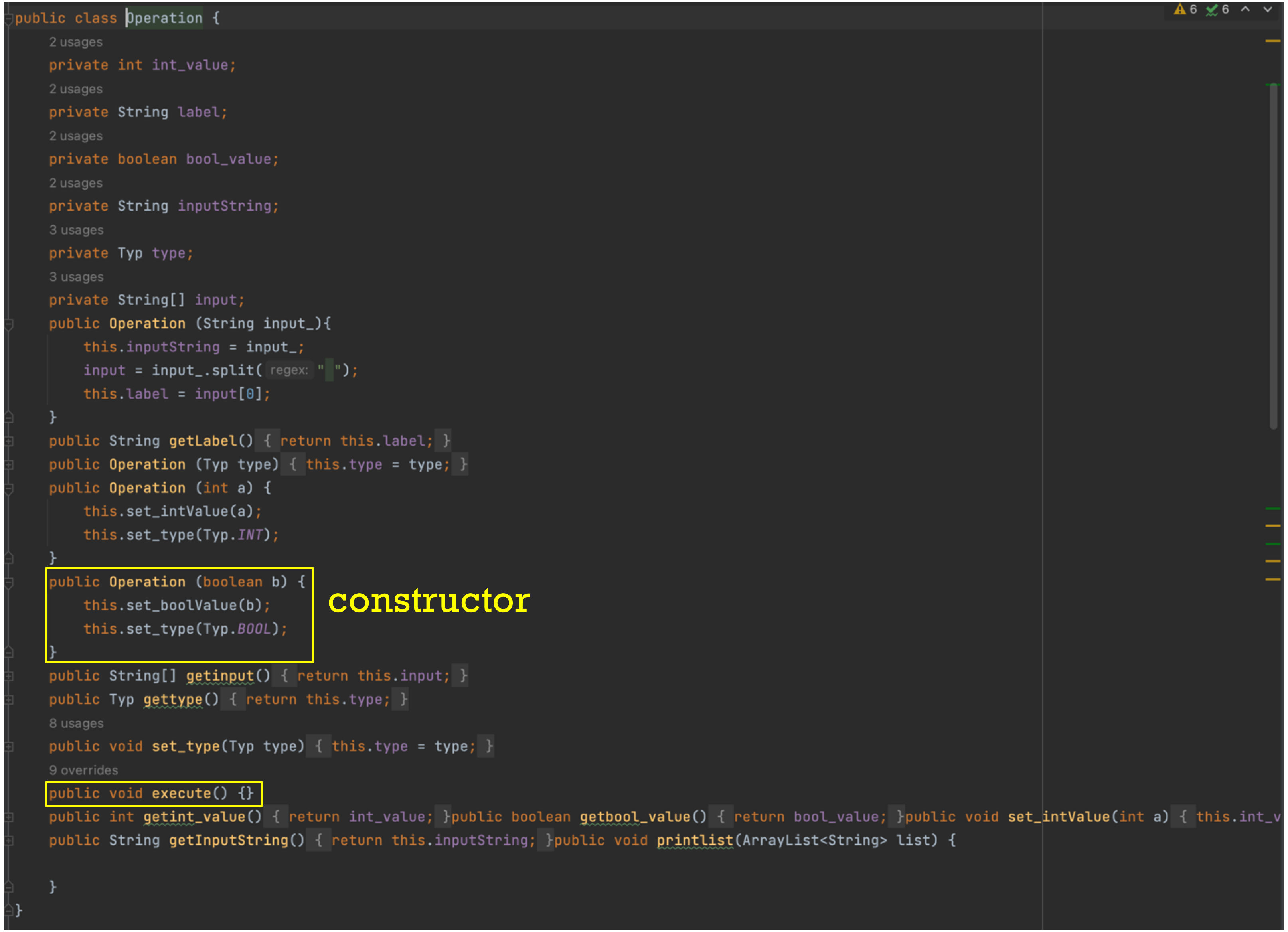
```
public static void listExpression(ArrayList<String> list,String expName) {
    if (setHashMap.getExpressionMap().containsKey(expName)) {
        if (((Operation) setHashMap.getExpressionMap().get(expName)).getLabel().equals("binexpr")) {
            ((binaryExp) setHashMap.getExpressionMap().get(expName)).printlist(list);
        }
        else if  (((Operation) setHashMap.getExpressionMap().get(expName)).getLabel().equals("unexpr")) {
            ((unaryExp) setHashMap.getExpressionMap().get(expName)).printlist(list);
        }
    }

}
```



```
public static void executeStatement(String name) {
    switch (((Operation) setHashMap.getStatementMap().get(name)).getLabel()) {
    case "vardef":
        ((varDef) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "assign":
        ((Assign) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "print":
        ((Print) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "skip":
        ((Skip) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "block":
        ((Block) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "if":
        ((IfElse) setHashMap.getStatementMap().get(name)).execute();
        break;
    case "while":
        ((While) setHashMap.getStatementMap().get(name)).execute();
}
}
```

REUSABILITY

- In the every function, we always need to use statement and decide its matched function.
- Than we create "executeStatement" and "listStatement" to use each statement to do the corresponding issues.



```
public class Operation {
    2 usages
    private int int_value;
    2 usages
    private String label;
    2 usages
    private boolean bool_value;
    2 usages
    private String inputString;
    3 usages
    private Typ type;
    3 usages
    private String[] input;
    public Operation (String input_){
        this.inputString = input_;
        input = input_.split( regex: " ");
        this.label = input[0];
    }
    public String getLabel() { return this.label; }
    public Operation (Typ type) { this.type = type; }
    public Operation (int a) {
        this.set_intValue(a);
        this.set_type(Typ.INT);
    }
    public Operation (boolean b) {
        this.set_boolValue(b);
        this.set_type(Typ.BOOL);
    }
    public String[] getinput() { return this.input; }
    public Typ gettype() { return this.type; }
    8 usages
    public void set_type(Typ type) { this.type = type; }
    9 overrides
    public void execute() {}
    public int getint_value() { return int_value; }public boolean getbool_value() { return bool_value; }public void set_intValue(int a) { this.int_v
    public String getInputString() { return this.inputString; }public void printlist(ArrayList<String> list) {

    }
}
```

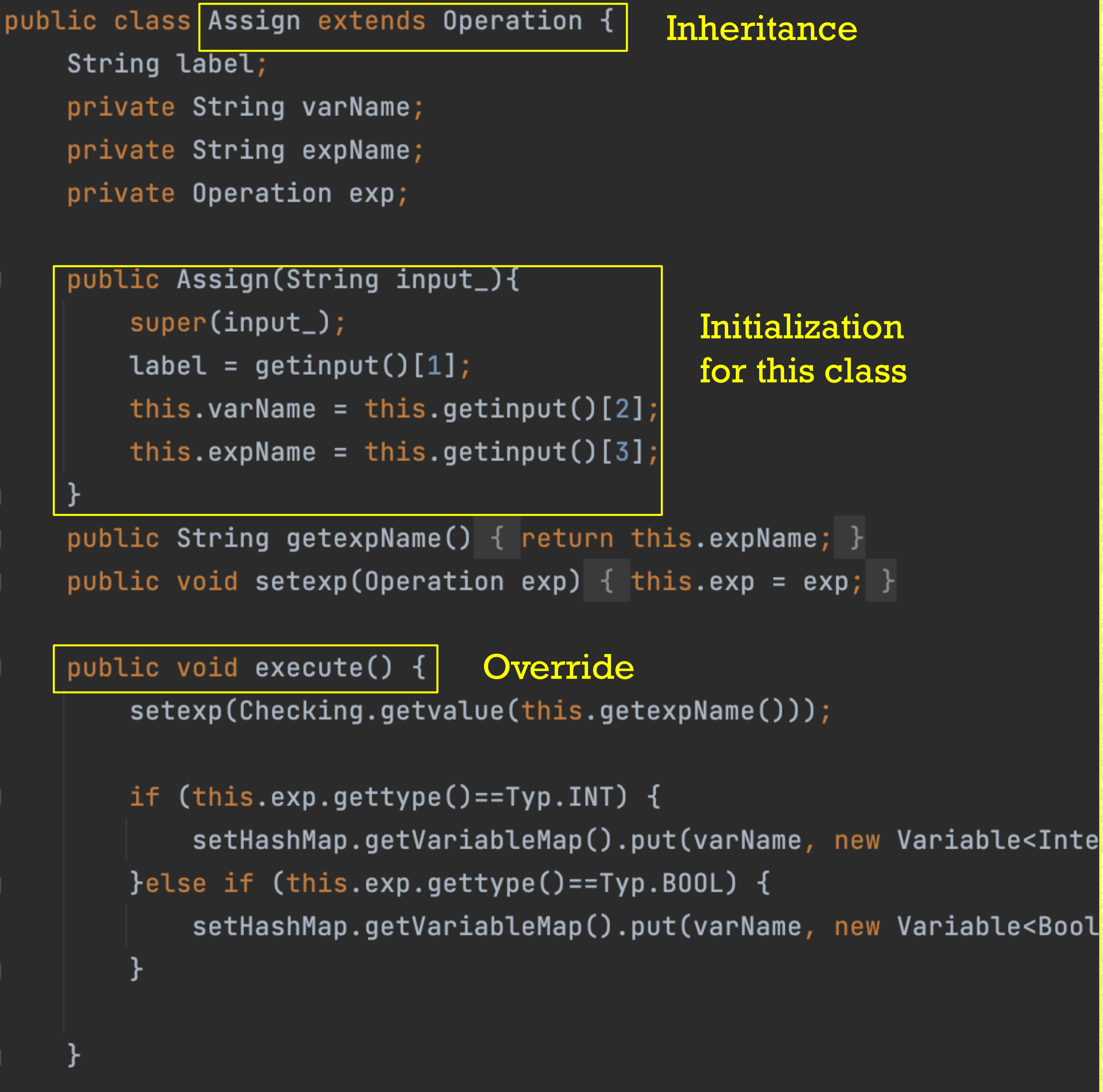
constructor

SCALABILITY

- This class is the superclass of the majority of the following function classes which will be seen in the required part.

- You will find that the function classes mostly extend this class and then do some changes or overrides based on this fundamental class.

SCALABILITY——EXAMPLE



```java
public class Assign extends Operation {
    String label;
    private String varName;
    private String expName;
    private Operation exp;

    public Assign(String input_){
        super(input_);
        label = getinput()[1];
        this.varName = this.getinput()[2];
        this.expName = this.getinput()[3];
    }
    public String getexpName() { return this.expName; }
    public void setexp(Operation exp) { this.exp = exp; }

    public void execute() {
        setexp(Checking.getvalue(this.getexpName()));

        if (this.exp.gettype()==Typ.INT) {
            setHashMap.getVariableMap().put(varName, new Variable<Inte
        }else if (this.exp.gettype()==Typ.BOOL) {
            setHashMap.getVariableMap().put(varName, new Variable<Bool
        }

    }
```

Inheritance

Initialization for this class

Override

THANKS